MDL FORTRAN STANDARDS

1.0 INTRODUCTION

The critical importance of developing well documented and well structured code has become more obvious with time. Except for, possibly, some small programs/subroutines written exclusively to test an idea or structure that will soon be discarded, Government developed software will be inherited and maintained by others. "Tricky" coding in the name of efficiency is to be avoided (although the definition of tricky will vary with individual).

It is imperative that we follow good coding and documentation rules in the development of all code, and in particular code that is to be handed off for use outside of MDL. Reasons include:

- Most development today involves more than one person. With several persons involved in a project, it is important that guidelines be followed so that all can easily "read" another person's program.

- Usually, it will fall to someone other than the originator to modify or maintain a program at some time in the future. Again, if a program has been written and documented according to prescribed rules, revisions and maintenance are much easier. This applies to external documentation as well as the code itself.

- Code developed by the DDTs is for the express purpose of implementation and integration into a much larger system. If all such code (including locally-developed code form the field) follows the same guidelines, understanding and dealing with it will be much easier, and we will be able to answer questions more readily than otherwise. Documentation will, of course, be mandatory.

- Standardization will reduce errors in coding and keystroking. The eye and mind become accustomed to "patterns," and a break in pattern may be an error. If there are no established patterns in the code, or if the patterns are considerably different from those to which the reader is accustomed, this human error detection feature cannot operate effectively.

- Converting a body of software from one computer system to another is easier if it is all written and documented to the same standards.

- Persons writing code and having it keystroked by others need not explain a preferred format; it will already be defined. Documentation may be assigned to a person other than the one writing the code; an established procedure makes individual coordination on a documentation format unnecessary.

- New employees with little or no programming experience can be more easily trained in good procedures if those procedures are written down and everyone follows them.

- Some simple optimization procedures, if followed, can reduce

execution time considerably. However, the primary purpose for these guidelines is not central processor optimization. Also, what is optimum for one system may not be for another.

In summary, the objectives of these guidelines are to enhance clarity, testability, maintainability, and person-to-person and computer-to-computer transferability of software throughout its life cycle.

Any system of software guidelines or standards is somewhat arbitrary. Different organizations have different standards, and textbooks do not agree. It is not so important exactly what the guidelines are, as it is that there be guidelines (assuming some semblance of reasonableness, of course).

This document contains coding guidelines for FORTRAN; a companion document contains guidelines for the C language.

2.0 FORTRAN CODING STANDARDS

The programming language to be used is the version of FORTRAN appropriate to the platform for which the code is intended. FORTRAN 90 should be used whenever available. Vendor-specific extensions to the FORTRAN 77 standard can be used when they conform to FORTRAN 90 standards; when they do not, they should not be used unless absolutely necessary.

Appendix I provides a code example to which the reader should refer when reading the following guidelines.

Documentation Block - Every program and subroutine must start with a documentation block following the outline in AIFM Appendix 1. Starting column convention is imposed to promote readability. Generally, in the absence of specific guidelines, standard typing rules should be used in preparing the documentation. If the system being used supports lower case characters as well as upper case, then it is optional which is used for the documentation block. Lower case for documentation does distinguish that material from executable code (which shall be upper case) but does add a degree of complexity and non-uniformity among programs/programmers.

Program Name - The first line should be the subroutine name starting in Col. 7. If it is a main program, and the compiler doesn't permit a program name, substitute a Comment statement with the program name.

Date, Programmer, Organization, Computer - Maintaining the exact date is not important; it is not used, for example, as the date the routine was added to the library. The month and year are sufficient. Starting in Col. 10, the date, the programmer's name, MDL, and the computer system the program was written for are each put on the third line, after a blank comment line, separated by three spaces. Extra lines should be used here to indicate modification dates, etc., as appropriate. Spacing may be adjusted to "line up" names, etc.

Purpose - Following another blank line, the next line should contain the word PURPOSE starting in Col. 10. Following that will be a short paragraph explaining the purpose of the routine. This need not be extensive, as details can be placed in the program writeup (external documentation). However, it should be complete enough to be useful to the user. If the routine was written specifically for a calling routine, the comment CALLED BY XXX is useful. Start all lines in this

paragraph in Col. 14.

Data Set Use - After the paragraph on "purpose" and a blank line, the next line should contain DATA SET USE starting in Col. 10. Listed below this line will be data set names followed by a brief explanation of them (see AIFM Appendix 1). The explanation should state whether the data sets are input, output, or internal. If no data sets are used by this routine, put NONE on the line following DATA SET USE.

Variables - The statement following those explaining data set use should contain the word VARIABLES starting in Col. 10. Following that, most, if not all, variables used in executable statements in the program should be defined in the format shown in AIFM Appendix 1. The equal sign should be in Col. 23 followed and preceded by one space. All lines except the one defining the variable start in Col. 25 unless some further indention seems appropriate, such as in lists. (Standardization here will allow copying from one routine to another when the variable is used in more than one routine. However, many times the explanation will have to be changed slightly for it to pertain to a particular routine.) Variables appearing only in COMMON need not be defined, but when a variable is used in COMMON and in other places in the routine, it must be defined. Variables used only to pass on to another subroutine should be defined, but is not mandatory. The cross reference list of the compiled source will identify where the variable is passed on.

List all variables in the subroutine call sequence, if any, first and in order. No other ordering is mandatory, but some logical sequence should be used and the best one to use may depend on the routine. The ordering might be alphabetical, especially if there are many variables. The order could be the approximate order the variables are first used in the program, especially the input variables; having the definitions of the input variables from an external source all in one place, and in order, has proven to be very useful. For each variable that is in the call sequence, place at the end of the comment either (INPUT), (OUTPUT), (INPUT-OUTPUT) or (INTERNAL) to indicate its use in the subroutine. (This is not appropriate for a main program.) This should also be done for variables actually used that are in COMMON. If, and only if, the type of variable is other than INTEGER*4 or REAL*4, place the type in parentheses at the very end of the comment, e.g., (CHARACTER*5).

Another option for grouping variables (other than those in the call sequence) is to have sections headed INPUT, OUTPUT, etc. (starting in Col. 14) and to put the appropriate variables under these headings.

Non-System Subroutines Used - The non-system subroutines used in the program are listed, separated by a comma and space and indented to Col. 14, following the section heading NON-SYSTEM SUBROUTINES USED, starting in Col. 10.

Declarative and Data Statements - Such statements, if any, should immediately follow the documentation block. An order such as PARAMETER, COMMON, TYPE, DIMENSION, EQUIVALENCE, and DATA is appropriate. Always use PARAMETER first, and DATA last.

PARAMETER - PARAMETER statements shall define a variable only where a DATA statement will not suffice, namely, in the definition of variable array dimensions or, rarely, when a computation is desired within the

definition to retain the computed formula. The cross-reference lists provided by some compilers do not treat variables defined with PARAMETER statements the same way as other variables, and some ignore them altogether; this makes checkout more difficult. This convention will let the user know that any variables defined in PARAMETER statements are variable dimensions.

COMMON Blocks - COMMON blocks should be used sparingly, if at all. Generally, code is easier to follow when the variables needed are passed through the call sequence rather than in COMMON, especially when some of the variables in the COMMON are used and some are not. Having variables in COMMON can also make it difficult to modify a program that has many subroutines. In any case, all COMMON should be labeled. The name of the block should be rather unique to keep to a minimum conflicts that might arise when a routine is used by others. For instance, XXXONE might be a good name for a program named XXX; this would better than BLOCK1.

Type Statements - Type statements should not be used unless the type is "unusual." The CHARACTER type is unusual in this sense and is needed for character variables. Do not use type statements for REAL*4 or INTEGER*4 variables. (See Variable Naming below.)

Variable Naming - The FORTRAN predefined specification of integer and real variables shall be followed--INTEGER(I-N), REAL(A-H,O-Z). By using this convention, it is much easier to catch integer/real conversion errors than if the reader has to remember the type of all variables in a specific routine. With the advent of FORTRAN 77, reserving the letter "C" for CHARACTER variables in new code is recommended. Do not use the IMPLICIT statement. For maximum portability, limiting the variable name to six characters is advisable, but not imperative. Variables used for only one purpose (e.g., to hold values of dew point temperature) should be given easily recognizable names (e.g., DEWP). (Using an array for multiple purposes may make this difficult, if not impossible, but equivalencing should not be used to overcome the difficulty.) Generally, the use of single characters, such as "I" and "J," should be reserved for DO loop indices. In two-dimensional grid indexing, the use of "IX" for left to right and "JY" for bottom to top is a good practice, and the convention of using the first index to refer to the "IX" direction is mandatory. When a variable is passed to another routine, whenever practical use the same name for the variable in both routines.

EQUIVALENCE Statements - Equivalencing variables tends to make code harder to follow, and encourages mistakes. It may also hinder optimization in some compilers. Only in special cases or where much memory can be saved should equivalencing be used, or where character information must occupy an INTEGER or REAL variable.

DATA Statements - When values are specified in DATA statements, try to arrange them so that they can be easily read. This is especially important for multiply dimensioned arrays. Put on separate lines whenever practicable values pertaining to different dimensions.

In-Line Documentation - In-line documentation should be provided at appropriate points in the program. Somewhere between 10% and 50% of the total lines should be devoted to documentation (besides the documentation block). The comments are used to explain the code and should be subordinate to it.

Therefore, with code that has executable statements starting in Col. 7, start all comments in Col. 10. One should expect to "read" the code with explanation by the comments, rather than vice versa. (Indention for IF THEN ELSE structures with accompanying comments will be treated later.) A block of code can be explained before the block by comments separated above and below by a blank line ("C" only on the line). A single line of code can be explained by a single comment following (or preceding) the executable line with no blank line. A comment should be used to explain the purpose of a called subroutine. Comments can be either upper or lower case, but the usage shall be consistent within a routine. Clarity is many times enhanced by inserting a blank line after a branch-type instruction. Comments are not to be put on the same line as an executable statement following an "!".

Length of Programs - Program (subroutine) length (number of lines of executable code) should be governed by the function of the routine, and not by some arbitrary rule such as "all programs will between 10 and 100 lines of code." A specific maximum size is not as important as convenient program structure. Modularity is important when meaningful, and it usually is.

Top Down Coding - Program flow should be from the top down. With the IF THEN ELSE type of structures of FORTRAN 77, this is always possible with enough nesting. It is usually possible to do this even when the GO TO construction is used. Some slight duplication of code may be preferable to branching. In all cases, it is the clarity of the code that is important. It may be confusing to have nests more than, say, 6 deep. On the other hand, if a program essentially repeats itself when input data so indicate, a branch from somewhere (usually near the end) back to (near) an input statement should not be confusing, and may be more "natural" than trying to accommodate this option with an IF THEN ELSE construction.

Statement Labels - Statement labels should always start in Col. 2, no matter how many digits they contain. Number only those statements to which reference is made (i.e., only those it is necessary to number). Most cross reference lists will indicate any statement numbers that can be removed.

Some logical numbering sequence must be followed. Some possibilities are:

> The numbers range from 1 through 9999 and be in sequence.
>
> The numbers always contain 4 digits and be in sequence.
>
> The primary numbering system start at 100 or above and end at 999, but, upon revision, when it is necessary to insert more numbers than space has been provided for, a fourth digit is added. Since all numbers start in Col. 2, they "appear" to be in order even though 1115 comes between 111 and 112 (this may be slightly inconvenient in some compiler's cross reference listings, as all 3-digit numbers may precede 4-digit numbers). Although this method may seem at first glance to be more complicated, it is really very simple and workable.

Statement Format - For programs that are basically not in the IF THEN ELSE structure, start all statements (except comments) in Col. 7. Continuation statements should be indented by at least 5 spaces unless there is a reason to do otherwise (a FORMAT statement can usually be split between lines with no problem--even a string of characters can be stopped and restarted on another line). Limit the line length to the FORTRAN 77 standard, 72 characters.

Statements should not include blanks unless they are necessary to improve readability. Establish a pattern and stick with it. Examples as used in MDL programs are:

```
            SUBROUTINE INTR(P,BY,BX,BB)

            DIMENSION SAVE(2,2),P(61,81)

            EQUIVALENCE (P(1,1),NPK(1)),(X,Y)

            COMMON/M400/VRBL1(10),VRBL2(10)VRBL3(100),
        1           VRBL4(1000)

            CALL RDMOSH(N,NWDS,NROWS,NCOLS,JDATE,NERR)

            WRITE(KFIL12,130)KDATE(MT),JDATE

 130  FORMAT(' THERE IS A PROBLEM WITH THE INPUT DATA NEEDED, KDATE ='I8,'. ',
     1       ' DATE FOUND IS ='I8)

            X=IB(J)+IA(K)+3*(K+IC(J)**4)+M/N

            CHARACTER*3 CWSFO,CNODE,CTIME(10)

            DATA NCRIT/2,1,1,1,1/

            PARAMETER (ND2=41,
        1          ND3=39)

            STOP 115
```

Note that a comment following an "!" shall not appear on the same line as an executable statement.

Continuation Lines - Continuation lines can be denoted by the sequence of numbers 1 through 9, then alphabetically starting with A. Occasionally, it may be desirable to start the sequence with 2 rather than 1 in DATA statements. As an option, the same character can be used for all continuation lines.

Spaces Versus Tabs - When spacing over to where a statement, statement label, or comment is to start, use the space bar, not the tab.

CONTINUE Statements - Continue statements should be used only where necessary, except a CONTINUE is always used at the end of a DO loop. End each DO loop with a separate CONTINUE statement even though this is not logically necessary. This serves the purpose of notifying the "reader" that this is the end of a DO loop, and may aid in optimization for some compilers. Each nest of a nested DO loop will have its own CONTINUE.

DO Loops - A blank comment should immediately precede a DO statement and follow the DO loop's CONTINUE statement. For very short, multiple nests, a separate blank for each loop is not needed.

FORMAT Statements - Format statements should be used in the code where they are referenced, and should be numbered in sequence along with other numbered lines. A FORMAT statement should immediately follow the first I/O statement

which refers to it. For ease of possible later modification, it may be best to duplicate a FORMAT statement, except for its number, so that it can be with the statement that refers to it. If multiple statements refer to the same FORMAT, later modification may remove (or renumber) the FORMAT, even though it is referred to elsewhere in the program, and a compile error will occur. When looking at the printed output and the code that produced it, it is much easier to match the output to the FORMAT statement when the FORMAT and the I/O statement are together.

Indention - Several rules for indention of statements are given above in connection with other topics. In general, when the GO TO structure predominates, start executable statements in Col. 7 and comments in Col. 10. For IF THEN ELSE structures, some indention shall be used. One option is to indent each "nest" another 3 spaces. Comments could be indented 3 more spaces. Whatever convention is adopted for a routine, it must be used consistently within the routine.

I/O Device Reference - Device reference by FORTRAN number should be with an INTEGER variable, not a constant. For main programs, this variable should be given a value in a DATA statement. For subroutines, this variable should be passed through the argument list, after being defined in the main program. In some cases, it may be more convenient to read the variable name from a control file. A convenient convention is KFIL1 for Unit No. 1, etc. A device reference number should always be passed to a subroutine to be used for the default output. A convenient name is KFILDO and if used consistently can be easily identified for that purpose in all routines.

Variable Dimensions - Whenever there is a chance that the dimensions of a variable will be changed, and always when the dimensions are referred to in other statements (for example, to keep from overflowing the array), the dimensions should be declared by defining a variable in a PARAMETER statement. The actual number should never be referred to in the code, but rather referred to by the variable name used in the PARAMETER statement. Usually, variables and their dimensions should be carried to subroutines through the argument list.

Subroutine Call Sequence - No matter what rules are established, exceptions will occur. Common sense must prevail. However, to the extent practicable, the order should be as follows:

- If data set reference numbers are provided, put them first.
- Other input to the routine should follow.
- Variables used for both input and output or work area should then follow.

  Output variables, ending with an error (return) code (if any) and finally the alternate return symbol(s) (return to a statement number--FORTRAN 77 uses an * for this purpose in the SUBROUTINE statement) should come last. Alternate returns should be used sparingly, as following the program logic is usually more difficult than using an error (return) code and checking it for desired branches. However, alternate returns are very useful in some situations (e.g., repeated calls to a subroutine where the same action is to be taken for all such returns).

Variable dimensions for an array or arrays should follow the last array name in which they are used. Multiple dimensions passed for an array should occur in the same sequence as they occur in the DIMENSION statement. For extensive call sequences, the dimensions could all be put together near the end.

Subroutine Entry Points - Each subroutine should have only one entry point. Do not use the ENTRY statement.

End of File and Error Checks - Error checks for input should be used. Errors can be indicated by an error code returned to the calling routine (preferably with a print--actually a WRITE to the unit KFILDO--of the diagnostic in the routine itself and with the value returned in the variable IER), or exit can be made to an error handling routine. In case the error is fatal, it may be all right to stop in the routine itself with an appropriate diagnostic (see Program Termination below).

Error Codes - Whenever possible, the "no error" condition should be "0." Use these as INTEGER variables, not as, for example, LOGICAL.

Non-Standard Features - Most compilers will permit use of some non-FORTRAN 77 features. Sometimes a FORTRAN 77 statement and its older counterpart (e.g., FORTRAN 66) may both work. When converting a program to FORTRAN 77, if the compiler does not flag the "error," it may go unrecognized; this is inevitable. However, we should stick to the FORTRAN 77 version as best we know it and can. System subroutines whose likelihood of being used by another compiler is not high should be avoided.

Indexing Variables with Multiple Dimensions - Whenever practicable, in nested DO loops, index the first variable indexed with the innermost DO. This is computationally more efficient for some compilers and may help to reduce paging in large, complex systems. It may be impossible to always follow this rule, but it shall be followed whenever it is reasonable to do so.

Program Termination - Generally, main programs should indicate in the normal print medium a successful completion, such as "XXX COMPLETED," where XXX is the program name. Any other stop should:

Produce in the print medium an indication of the problem and where the stop occurred. The latter can be done by using a statement such as "STOP AT 1013" where 1013 is the statement number at or near where the stop occurred. The termination of the program should be with the statement STOP 1013.

If the stop is in a subroutine, an error statement which includes the subroutine name should be printed, such as "STOP IN XXX AT 1013" where XXX is the subroutine name and 1013 is the statement number at or near where the stop occurred. The reason for the stop with values of pertinent variables should also be printed if such would be helpful. The termination of the subroutine should be with the statement STOP 1013. A little time spent arranging for this diagnostic may save much time later on.

Generally, it is better to return an error code from a subroutine rather than to terminate when there is a problem; this should be done if the user can exercise judgment about how to proceed. However, if the error is unrecoverable, or if in the judgment of the author of the subroutine

it would definitely be a mistake to continue, the stop can be in the subroutine. (The best procedure to use may vary with circumstance; a STOP in a subroutine may be prohibited in "operational" jobs.) In any case, the user must always be protected from bad results or data.

Printed Output - Output to be printed should be arranged in an easy-to-read format. For instance, line up columns of numbers. Also identify values printed in well-understood terms or in terms of variables defined in the program writeup.

3.0 REVISION OF EXISTING CODE

Much of the development done centrally and locally will consist of revision of existing code gathered from various organizations. Needless to say, this code will come in varying forms of completeness and documentation. When the decision is made as a group effort as to which code to use, it will be determined as to how much to make the code conform to the guidelines. For extensive rewrites, it may be advisable to make it roughly conform. However, for more minor changes, it will be better to follow the "style" of the existing code (provided, it has a consistent style). Two conflicting goals can provide some guidance: (1) Use existing code as much as possible, and (2) make sure each module is well structured and documented, as well as being reasonably efficient and fulfilling the required functions. Note that it is not necessary to remove all "GO TOs" to achieve optimum metrics.